AD-A059 861 NAVAL OCEAN SYSTEMS CENTER SAN DIEGO CA F/G 9/2
PASCAL 1100: AN IMPLEMENTATION OF THE PASCAL LANGUAGE FOR UNIVA--ETC(U)
JUL 78 M S BALL

UNCLASSIFIED NOSC/TR-271 NL

1 OF 1

AD
A059861

END
DATE
FILMED
12-78
DDC

(12) LEVEL II

NOSC

(14) NOSC/TR-271

Technical Report 271

(6)

PASCAL 1100: AN IMPLEMENTATION OF THE PASCAL LANGUAGE FOR UNIVAC 1100 SERIES COMPUTERS.

(10) MS Ball

(11) 1 Jul~~y 1~~978

(rept.,)

(9) Research and Development: Oct~~ober 1~~976 – Sept~~ember 1~~977,

Prepared for
Naval Sea Systems Command

(12) 26p.

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

NAVAL OCEAN SYSTEMS CENTER
SAN DIEGO, CALIFORNIA 92152

393 159

NAVAL OCEAN SYSTEMS CENTER, SAN DIEGO, CA 92152

AN ACTIVITY OF THE NAVAL MATERIAL COMMAND

RR GAVAZZI, CAPT. USN
Commander

HL BLOOD
Technical Director

ADMINISTRATIVE INFORMATION

This report is a revised and expanded version of Naval Undersea Center Technical Publication 527 of September 1976. The original work was sponsored jointly by the Naval Sea Systems Command (code 06H1) and Naval Electronic Systems Command (code 310). The revision was sponsored by the Naval Sea Systems Command (code PMS 406) as part of the Advanced Lightweight Torpedo Program.

Released by
RH HEARN, Head
Electronics Division

Under authority of
DA KUNZ, Head
Fleet Engineering Department

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
NOSC TR 271

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
PASCAL 1100: AN IMPLEMENTATION OF THE PASCAL LANGUAGE FOR UNIVAC 1100 SERIES COMPUTERS

5. TYPE OF REPORT & PERIOD COVERED
Research and Development
October 1976-September 1977

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
MS Ball

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Naval Ocean Systems Center
San Diego, California 92152

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Naval Sea Systems Command
Washington, DC

12. REPORT DATE
1 July 1978

13. NUMBER OF PAGES
24

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes the features of Pascal language peculiar to its implementation on Univac 1100 series digital computers. It defines terms left undefined in the general language and describes local extensions to the language. It also describes procedures for using the language under the 1100 Executive.

CONTENTS

INTRODUCTION . . . page 3

1 HARDWARE REPRESENTATION . . . 3

2 EXTENSIONS TO THE LANGUAGE . . . 4

- 2.1 Processor Format Call . . . 4
- 2.2 External Procedures . . . 5
- 2.3 FORTRAN Procedures and Functions . . . 6
- 2.4 Enhancements of "Read" and "Write" . . . 7
- 2.5 Enhancements to "Reset" and "Rewrite" . . . 8
- 2.6 Octal Numbers . . . 8

3 SPECIFICATIONS LEFT UNDEFINED . . . 9

- 3.1 Program Heading and External Files . . . 9
- 3.2 Standard Types . . . 9
- 3.3 Standard Procedure "Write" . . . 10

4 RESTRICTIONS . . . 11

5 ADDITIONAL PREDEFINED IDENTIFIERS . . . 11

- 5.1 Constants . . . 11
- 5.2 Variables . . . 12
- 5.3 Procedures . . . 12
- 5.4 Functions . . . 14

6 BRINCH HANSEN PASCAL . . . 14

- 6.1 Extensions for Compatibility . . . 14
- 6.2 The "B" Option . . . 15
- 6.3 Incompatibilities . . . 16

7 HOW TO USE PASCAL 1100 . . . 16

- 7.1 Compiler Options . . . 16

8 PERFORMANCE . . . 18

- 8.1 Compiler Performance . . . 18
- 8.2 Compiled Code Performance . . . 18

9 IMPLEMENTATION DETAILS . . . 19

- 9.1 Form of Generated Code . . . 19
- 9.2 Run-Time Organization . . . 20
- 9.3 Diagnostic System . . . 23
- 9.4 Representation of Files . . . 23
- 9.5 Miscellaneous Topics . . . 24

INTRODUCTION

This report describes the features of the Pascal implementation for the Univac 1100 series of digital computers. The Pascal language in general is described in the book

> PASCAL – User Manual and Report, by Kathleen Jensen and Niklaus Wirth, Springer-Verlag, New York, 1974 (Lecture Notes in Computer Science 18).

The present report describes only those features of the language that are peculiar to the Univac implementation.

The Pascal compiler is also capable of compiling programs written in a Pascal dialect used by Per Brinch Hansen in his minicomputer-based implementation of the language. Those features related to this implementation are always available to the user but are not generally available on other Pascal implementations. They will be described in a separate chapter.

This note is divided into nine sections as follows:

1. Hardware representation.
2. Extensions to the language.
3. Specifications left undefined.
4. Restrictions.
5. Additional predefined identifiers.
6. Brinch Hansen Pascal.
7. How to use Pascal 1100.
8. Performance.
9. Implementation details.

1 HARDWARE REPRESENTATION

Pascal 1100 uses the American Standard Code for Information Interchange (ASCII) character set exclusively. Within the compiler, lower case characters are converted into upper case before use as reserved words and identifiers, so that there is no difference between "begin" and "BEGIN" as far as the compiler is concerned.

All reserved words are written out without escape characters or underlining; for example,

begin, end, case ...

Only the first twelve characters of any identifier are significant, and identifiers which differ only after the first twelve characters are considered identical.

The special symbols given in the manual are used as given, with the alternates shown in table 1.1. All the alternates are in addition to the standard and are included for compatibility with other implementations and the keypunch character sets.

The character "_" (underline) is considered to be an alphabetic character within an identifier. The arrow, used for pointers, is represented by "^".

TABLE 1.1. ALTERNATE SYMBOL REPRESENTATIONS.

Standard	Alternate
and	&
{	(*
}	*)

2 EXTENSIONS TO THE LANGUAGE

This section describes nonstandard language constructs available on the Pascal 1100 system. They are oriented toward the 1100 operating system and exist primarily to allow utilization of the operating system facilities.

2.1 PROCESSOR FORMAT CALL

There are two basic formats for executing a program under the 1100 executive. The most commonly used method for user programs is

```
@XQT,OPTS PROGRAM
```

This is the format with which a normal Pascal program should be executed.

The second format, used for processors and control cards, is

```
@PROGRAM,OPTS ARG1,ARG2,...
```

This method has the advantage of allowing the user to specify input and output files to the program on the calling card. It is also the format expected by the processor interface routines. A Pascal program may be constructed for this method of calling by replacing the program heading by the following

⟨program⟩ ::= ⟨program heading⟩ ⟨block⟩ . | ⟨processor heading⟩ ⟨block⟩ .

⟨processor heading⟩ ::= PROCESSOR ⟨identifier⟩ (⟨process parameters⟩) ;

⟨process parameters⟩ ::= ⟨file specifier⟩ { , ⟨file specifier⟩ }

⟨file specifier⟩ ::= ⟨identifier⟩ | ⟨identifier⟩ *

The ⟨processor heading⟩ generates a program which can be called using the second format above. The ⟨file specifier⟩ operates as usual, except that an asterisk (*) following "INPUT" instructs the program to take its input from the system source input routine (SIR). If the input is from SIR, then the user also has access to the system routines for source output and relocatable output. This will be discussed later.

The INFOR table from the calling line may be manipulated using external procedures available in the library. See the Univac Programmer's Reference Manual (PRM) for further data.

For example:

```
PROCESSOR prtd(input*,output);
```

specifies that the program will be called by a card

```
@PRTD FILE,ELT
```

and will receive its input from the standard source input routine.

2.2 EXTERNAL PROCEDURES

The Pascal 1100 system allows the user to define and use externally compiled procedures. The declaration for an external procedure consists of a procedure heading followed by the word "EXTERN".
For example:

```
PROCEDURE printit(tab: integer; ch: char); EXTERN;
FUNCTION gcd(x, y: integer): integer; EXTERN;
```

The user can define procedures to be called in this manner by inserting the reserved word "ENTRY" after the "PROCEDURE" or "FUNCTION" in the procedure heading.
For example:

```
PROCEDURE ENTRY printit(tab: integer; ch: char);
    ⟨body⟩
FUNCTION ENTRY gcd(x, y: integer): integer;
    ⟨body⟩
```

Any procedure in the outer scope of the program may be declared an entry. In addition, a procedure or group of procedures may be compiled by themselves without a containing program. In this case, any variables declared outside the procedures are available for use in the procedures and will be retained from call to call. File variables may not be declared in this outer scope, since they will not be initialized properly. A group of such procedures must be terminated with a period (".") just as a program is terminated.
For example:

```
VAR i: integer;
PROCEDURE ENTRY initialize;
    BEGIN
    i := 0;
    END;
PROCEDURE ENTRY increment;
    BEGIN
    i := i + 1;
    END;
```

```
FUNCTION ENTRY current_value: integer;
    BEGIN
    current_value := i;
    END;
 {This period is necessary to terminate compilation}
```

There is a limited run-time check for proper call of external procedures. This checks that the procedure is called with the proper number of arguments (mod 16) and that the types and parameter modes of the first seven arguments are correct. The check is not very detailed and will not, for instance, find a call with an array of a different size. It will, however, catch most simple errors, such as an argument missing or the order of two arguments inverted.

There is no way to disable this check.

2.3 FORTRAN PROCEDURES AND FUNCTIONS

FORTRAN procedures or functions can be declared in a manner similar to external procedures. To use a FORTRAN procedure, the user declares it in the same manner as a Pascal procedure but replaces the body of the procedure with the word "FORTRAN". For example:

```
PROCEDURE plot(x, y: real; indx: integer); FORTRAN;
```

The compiler will generate the FORTRAN calling sequence for this procedure. Only code generated by the ASCII FORTRAN (FTN) processor may be used.

Although FORTRAN makes no distinction between value and variable parameters and will happily assign a value to an expression or constant, Pascal makes such a distinction and will make a local copy of any parameter declared to be value. This is done for safety reasons, and the programmer is advised to make use of this property whenever possible.

There are some incompatibilities in the representation of variables between Pascal and FORTRAN. The major one is the storage of arrays. Pascal considers multidimensional arrays to be arrays of arrays, which automatically defines storage with the rightmost of any group of subscripts varying more rapidly. This is called "rowwise storage." FORTRAN, on the other hand, uses columnwise storage, with the leftmost subscript varying most rapidly. This means that the user must transpose array elements in the Pascal code, or the FORTRAN program must take this into account.

Table 2.1 summarizes the type correspondence.

TABLE 2-1. FORTRAN vs. PASCAL TYPES.

Parameter type in FORTRAN	Parameter type in Pascal	Remarks
INTEGER	integer	
REAL	real	
DOUBLE	none	double precision not supported in Pascal

COMPLEX	record
	re; real;
	im: real
	end;	

LOGICAL	Boolean	
ARRAY	ARRAY	see the note above for incompatibility
SUBROUTINE	PROCEDURE	formal procedures must be FORTRAN procedures
FUNCTION	FUNCTION	as procedure, also, result cannot be COMPLEX or DOUBLE, as record-valued functions are not allowed

Only the above parameter types make sense if the external routine is actually written in FORTRAN. Note that the Pascal compiler does not check for legal argument types.

2.4 ENHANCEMENTS OF "READ" AND "WRITE"

The standard procedures "read" and "write" have been enhanced to work with all files, not just text files. The following equivalences now hold independently of file type.

```
write(f,v); ⟨==⟩ f^  := v; put(f);
read(f,v);  ⟨==⟩  v := f^; get(f);
```

In line with these definitions, "read" will take a component of a packed structure as an argument. This makes "read" an unusual procedure, but the formal equivalence given in the definition will hold.

A further enhancement is the addition of octal editing when writing numbers to a character file. Octal editing is specified by writing the word "OCT" after the editing statement:

```
write(f:4 OCT);
```

Octal editing is legal with all scalar or pointer variables and will write exactly the number of digits specified. If the value will not fit in the number of digits specified, it will be truncated from the left to fit. Any leading zeros will also be printed.

For example:

write(1024B:3 OCT); writes "024"

2.5 ENHANCEMENTS TO “RESET” AND “REWRITE”

The standard procedures “reset” and “rewrite” have been enhanced to allow a better interface to the operating system. They now take an additional two arguments, disk address and file name. These arguments are optional and may be left out if they are not needed. The declarations are as follows:

```
TYPE identifier = PACKED ARRAY [1..12] OF char;
PROCEDURE reset(VAR f: file; address: integer; fname: identifier);
```

Address, if specified gives the sector address within a mass storage file to which the file will be reset. It is the responsibility of the caller to ensure that this address is valid. “Fname,” if specified, causes the procedure to close out the current file and reopen the file with the internal name of the “fname.” This file must be assigned to the run at the time of the call.

For example:

```
reset(f,12);
```

resets the file f to sector address 12.

```
rewrite(f,12,'OUTFILE');
```

closes any file currently in use, attaches the file with the internal name 'OUTFILE', and resets to sector 12 of that file.

In addition, the new standard procedure “close” has been added to explicitly close a file. Its declaration is

```
PROCEDURE close(VAR f: file; VAR addr: integer);
```

where “addr” is an optional integer variable which will contain the next available mass storage address in the file at the time of closing.

The combination of these enhancements allows the user to deal with 1100 executive element files directly, as well as attaching and freeing user files dynamically from within the run.

2.6 OCTAL NUMBERS

The user may specify octal numbers to the compiler by following an integer with the character “B”. Thus 1000B is equivalent to 512.

3 SPECIFICATIONS LEFT UNDEFINED

3.1 PROGRAM HEADING AND EXTERNAL FILES

A Pascal file variable is implemented as a file under the 1100 executive. External files are those specified in the program heading and must be assigned to the run at the time of program execution. Internal files are generated on entry to the procedure in which they are declared and are released upon exit from that procedure.

The following predeclared files may be used in the program heading and allow access to 1100 executive standard files.:

Input:text; This is the standard input file maintained by all programs. It is the file obtained by the normal system "READ$" routine. To allow the use of Pascal programs with demand terminals, this file is initialized with EOLN true. To obtain the first image, the user must issue a "read." A "reset" of this file has no effect. The maximum line length allowed is 132 characters.

Input*:text; When followed by an asterisk, this file obtains its data from the source input routine. In this case, a "reset" will close out the current pass and begin the next pass over the input data. Since the user cannot use the source or relocatable output routines until after the first pass is completed, this is quite important. In all other ways, the performance is identical to the normal use of "input." See the Univac PRM for more data. The maximum line length allowed is 132 characters.

Output:text; The output file writes to the standard output stream using the system "PRINT$" command. It must be declared in any program. A call to "rewrite" has no effect on this file. The maximum line length allowed is 132 characters.

SOR:text; This writes a symbolic element using the system symbolic output routine (SOR). This is meaningful only when a processor call card is used. The file must be initialized with a "rewrite" after the first pass of SIR (if used) is complete. Subsequent "rewrites" to this file will produce an error termination. The maximum line length allowed is 132 characters. See the Univac PRM for more data.

Punch:text; This produces punched cards using the system "PUNCH$" command. A call to "rewrite" will have no effect. Note that the maximum record length for this file is 80 characters. The ASCII characters written to this file will be converted to their Fieldata equivalents before punching.

3.2 STANDARD TYPES

3.2.1 INTEGER

The standard type "integer" is implemented with the standard 1100 series interger word, which allows values in the range -34359738367..34359738367.

Warning: To check for integer overflow, except on divide, is impractical on the Univac 1100 series. It is the user's responsibility to make sure that these limits are not exceeded.

In line with the above, the standard identifier "maxint" has the value 34359738367.

3.2.2 REAL

Real numbers are implemented using the 1100 single-precision floating point format. This allows 27 bits for the fraction, which corresponds to about 8 significant decimal digits. The values of the exponent are between –39 and 38. Exceeding the upper limit causes a run-time error, and dropping below the lower limit causes the resuit to be set to a true zero.

3.2.3 CHAR

The type "char" contains all of the standard ASCII characters, including unprinting control characters. Although ASCII is a 7-bit code, with possible ordinate values between 0 and 127, the 1100 series typically allows 9 bits for each character and stores them 4 to a word. The Pascal system also follows this practice, and a packed array of "char" will have 4 characters per word, each with 9 bits of space allocated.

3.3 STANDARD PROCEDURE "WRITE"

If no minimum field length parameter is specified, the default values shown in table 3.1 are assumed. If the length of a line will exceed the allowed length for the file (80 for punch, 132 otherwise), the line will be terminated and a new line written. The effect is the same as if "writein" had been called. There are no spacing control characters at the start of each line, and all the characters in a line are printed.

The standard data format for text files truncates trailing blanks, then pads the number of characters to a multiple of 4 with blanks. This means that if a file is written and then read, the number of blanks read at the end of the line may not be the same as the number written. (What can I say?)

TABLE 3.1. DEFAULT VALUES.

Type	Default
integer	12
real	12 (where the exponent is always written E+xx)
Boolean	12
char	1
a string	length of the string
octal format	12

4 RESTRICTIONS

The following restrictions must be observed in using the Pascal 1100 language:

1. The words "entry", "processor", and "univ" (used by Brinch Hansen style programs) are reserved.

2. The base type of a set must be as follows:

   a. A scalar with at most 144 elements (including "char").

   b. A subrange with a minimum element greater than or equal to 0 and a maximum element less than or equal to 143.

3. Standard functions or procedures cannot be used as actual procedure parameters. For instance, to run program 11.6 from the manual, one would have to write auxiliary functions as follows:

```
...

function sine(x: real); real;
     begin sine := sin(x) end;

function cosine(x: real); real;
     begin cosine := cos(x) end;

function zero(function f: real; a,b; real); real;
     begin ... end;

...

begin
     read(x,y); writein(x,y,zero(sine,xy));
     read(x,y); writein(x,y,zero(cosine,x,y));
end.
```

4. Formal procedures and functions must have only value arguments. This will be diagnosed at run time. Also, formal procedures and functions may not have procedure or function arguments.

5. It is not possible to construct a file of files; however, records and arrays with files as components are allowed.

6. It is not possible to declare files in a dummy outer block containing external procedures, as the files will not be initialized properly.

7. Files must be initialized for writing by calling "rewrite".

Thus it is not possible to read to the end of an existing file and then extend it by writing.

5 ADDITIONAL PREDEFINED IDENTIFIERS

5.1 CONSTANTS

The constant "linenumber" is predefined. This is an integer whose value is always the current line number in the source program. Although the value changes, the identifier is treated as a constant by the compiler. This is useful in debugging for inserting error messages which refer to the source line.

5.2 VARIABLES

The variable "options" is defined by

options: set of char;

This variable is allocated in every program and is initialized by the system using the options specified on the @XQT or the processor call line. The character corresponding to each option character will be included in the set. Note that the system considers all options to be upper case, no matter what is typed.

5.3 PROCEDURES

Additional predefined procedures include the following:

halt(message); Writes the string "message" to the print file and initiates an error walkback. This is primarily for use in library procedures written in Pascal.

mark(p); Sets the pointer "p" (which may be any pointer type) to the current heap top. This may be used with the procedure "release" below.

release(p); Resets the current heap top to the value in the pointer "p" (which may be of any pointer type). This value should have been set by the procedure "mark" above. These are primitive routines which allow the user to simulate a second stack on the heap. Since the routine "dispose" is presently not functional, this is the only method available to return storage to the heap.

close(file,addr) Closes the file specified just as if the block containing that file had been exited. If addr is included (it is optional), it will be set to the next available mass storage address in the file.

The following procedures are for programmers interfacing closely with the executive and are of little interest to the average programmer. In all cases, further details are available in the Univac PRM.

sror(kbits); Applicable only for processor. Opens the system relocatable output routine (ROR) for the generation of relocatable code. The integer "kbits" sets the "kbit limit" for the element.

ror(pkt); Applicable only for processor. Writes a word to the relocatable output. "Pkt" is any array which contains the relocation data and word to write.

eror(trans,transic); Applicable only for processor. Closes ROR and sets up the transfer address. "Trans" and "transic" are the relative address and location counter of the transfer location. If no transfer location is desired, "trans" should be negative.

tblwr(table,length); Applicable only for processor. Writes a relocatable preamble in the array "table" of length "length" to the relocatable element. ROR must be closed. For more data, see the Univac PRM.

er(index,"A0","A1","A2");

(Warning: Use of this procedure may be hazardous to your sanity.) This procedure is very low level and provides no protection for error. It does allow the user access to the 1100 executive "ER" mechanism from Pascal, an ability which is sometimes necessary. The "index" is a constant which is the ER index. It is checked against an internal set of allowed values before the call is generated. "A0", "A1", and "A2" are optional integer variables which, if they exist, are loaded into the hardware registers A0, A1, and A2 before the call. On return from ER, the final values of A0, A1, and A2 are stored in these variables. The function "address", described below, allows the use of this procedure with Pascal variables for the packets. Allowed ER index values are given in table 5.1.

TABLE 5.1. ALLOWED ER INDEX VALUES.

Name	Value	Name	Value
ABORT$	012B	ACLIST$	141B
ACSF$	140B	ACT$	147B
APCHCN$	075B	APRTCN$	074B
COND$	066B	DACT$	150B
DATE$	022B	EABT$	026B
ERR$	040B	ERRPR$	202B
EXIT$	011B	EXLINK$	173B
FACIL$	114B	FACIT$	143B
FITEM$	032B	FORK$	013B
INFO$	116B	IO$	001B
IOARB$	021B	IOW$	003B
LABEL$	031B	LINK$	171B
NAME$	146B	NRT$	062B
PCT$	064B	PFD$	106B
PFI$	104B	PFS$	105B
PFUWL$	107B	PFWL$	110B
RLINK$	172B	RLIST$	175B
RSI$	112B	RT$	061B
SETC$	065B	TDATE$	054B
TIME$	023B	TSWAP$	135B
TWAIT$	060B	WAIT$ *	006B
WANY$	007B		

* Because of the unusual calling sequence of WAIT$, it generates:

```
L     A0,"A0"
TP    3,A0
ER    WAIT$
S     A0,"A0"
```

5.4 FUNCTIONS

The following additional predefined function is available:

conv(j); Returns a real value for the integer "j". For use in Brinch Hansen Pascal, which has no implicit conversion to real.

The following functions are primarily for programmers writing code which interfaces closely with the executive or the hardware and should be of little interest to the average programmer. More details are available in the Univac PRM.

expo(x); Returns an integer equal to the exponent part of the real number "X". This is in excess 64 notation.

address(x); (Warning: Use of this procedure may be hazardous to your sanity!) Returns an integer value equal to the machine address of the variable "x", which may be of any type. This is provided to allow use of the standard procedure "er" and is otherwise of no use.

6 BRINCH HANSEN PASCAL

The compiler will accept the dialect of Pascal used by Per Brinch Hansen and called "Sequential Pascal" by him. For a complete description, see his publication:

> Sequential Pascal Report, by Per Brinch Hansen and Alfred C. Hartman, Information Science, Calif. Inst. Tech., July 1975.

The Pascal 1100 implementation makes no attempt to follow the restrictions of the Brinch Hansen dialect, but the following extensions allow the compilation of programs written in it. There is no guarantee that programs which run under this system will also be legal under Brinch Hansen's system.

6.1 EXTENSIONS FOR COMPATIBILITY

The following extensions are available at all times:

1. The symbols shown in table 6.1 may be used as alternate representations of standard symbols.

2. The reserved word "UNIV" is allowed in parameter lists. The modified syntax is

```
⟨parameter group⟩ ::= ⟨identifier⟩ {, ⟨identifier⟩} : ⟨parameter type⟩
⟨parameter type⟩ ::= ⟨type identifier⟩ | UNIV ⟨type identifier⟩
```

The use of "UNIV" before the type identifier informs the compiler that any parameter type is acceptable as long as it has the same size as the formal type. This is obviously highly machine dependent.

3. The procedure "conv" converts integer to real explicitly.

4. The boolean operators "and" and "or" will accept set operands. When they are used, "and" is equivalent to "*" and "or" is equivalent to "+".

5. A constant string of any length may be used as a value parameter or in an assignment statement to any packed array of characters. The lower subscripts are automatically aligned, and the string is truncated or extended with blanks to match the length of the array.

6. Within a string or character constant the syntax

⟨ordinate expression⟩ ::= (: ⟨number⟩ :)

may be used. This inserts into the string at that point a character whose ordinate is the value of ⟨number⟩.

7. Within an expression or a "WITH" statment, a pointer-valued function may be used in place of a pointer variable. For example:

```
type realp = ^real;
FUNCTION p(x: real): realp;
...
w := p(3.0)^ + 5.0;
```

This is forbidden by the Pascal report and Brinch Hansen's own report but used extensively in his code.

TABLE 6.1. ALTERNATE SYMBOL REPRESENTATIONS.

Standard	Alternate
[	(.
]	.)
^	@
{	"
}	"

6.2 THE "B" OPTION

The "B" option to the compiler (see the chapter on compiler use) specifies that the program being compiled is to be treated as a Brinch Hansen program. This makes the following changes in the form of the source program and the compiled code.

1. A "prefix" is now allowed. This contains type, constant, and procedure/function declarations. The type and constant declarations are entered into the outer scope of the program, and the procedures and functions are declared as external. This allows the direct simulation of Brinch Hansen's operating system interface through the prefix.

2. The main program will be compiled as an entry procedure which can be called from another program. In this case, the program argument list can have the same form as a procedure parameter list, rather than containing external file names.

3. All arrays of characters are treated as packed arrays.

6.3 INCOMPATIBILITIES

There are many incompatibilities between Pascal 1100 and the Brinch Hansen compiler. The user is referred to the publication above. The following are a few of the less obvious problems.

1. Brinch Hansen style compiler options are not accepted by Pascal 1100.

2. In Brinch Hansen Pascal, a case selector value with no corresponding statement is ignored. In Pascal 1100, it produces a run-time error.

3. Pascal 1100 does not initialize pointers to nil.

4. Brinch Hansen Pascal uses constant parameters, while Pascal 1100 uses value parameters. The difference is that no value can be assigned to a constant parameter within a procedure, while a value parameter can be treated as a local variable.

7 HOW TO USE PASCAL 1100

The Pascal system consists of a compiler, which converts the Pascal code into Univac 1100 series relocatable code, and a run-time library, which provides utility and input-output functions for the compiled code. The location of the compiler and run-time library is dependent upon the local system configuration. Check with the computer center for your site.

The Pascal compiler (PAS) is a Pascal program. It is a processor and is called in the standard manner. The processor call statement is

```
@PAS,OPTS SOURCE,RELOC,UPDATEDSOURCE
```

When the program is compiled, the relocatable elements must be collected with the Pascal run-time library, and the program can then be executed in the usual manner. Any external files must be assigned at the time of execution.

7.1 COMPILER OPTIONS

The behavior of the compiler may be varied by the use of certain options. There are two types of options recognized by the Pascal compiler: control card options, specified on the calling control card, and compiler directives included in the code.

A compiler directive is written as a special form of comment with a $-character as the first character

```
{$⟨option sequence⟩ ⟨any comment⟩}
```

For example:

```
{$S+,T- this sets "S" and resets "T"}
```

Normally, a "+" following an option will activate that option and a "-" will deactivate it.

7.1.1 COMPILER DIRECTIVES

The following options operate as compiler directives:

A Generate code to check the values assigned to variables of subrange type to make sure that they are within bounds.

default = A+

B Generate a Brinch Hansen style program

default = B-

E Generate code for machines other than the 1110. This causes the generated code to consider the possibility of the real residue affecting registers in use. See the section on "Non-1110 Code Generation" for details.

default = E-

L Generate a full compiler listing including generated code.

default = L-

R If one or two digits follow the R, this is the amount of extra space allocated for dynamic variables (those not declared in the main program). The number specifies this in thousands of words.

default = R2 2000 extra words

S Produce a source listing.

default = S-

T Include run-time tests for subscripts out of range and invalid pointer references.

default T+

Z Include line number diagnostics for error termination.

default = Z+

Most of these compiler directives can be activated and deactivated as often as required and can be applied selectively to different portions of the code. "B" and "R" are obvious exceptions.

7.1.2 CONTROL CARD OPTIONS

Control card options follow:

B	Set B+. See above.
E	Set E+. See above.
I	Inserting a new element from the run stream.
L	Set L+. See above.
O	Set A-, T- (omit tests). See above.
S	Set S+. See above.
Z	Set Z-. See above.

8 PERFORMANCE

The following performance was measured on a Univac 1110. All times given are totals, including both Central Arithmetic Unit (CAU) time and Control Card and Executive Reference (CCER) time.

8.1 COMPILER PERFORMANCE

The compiler's performance was measured as it compiled itself. The compiler consists of 7,494 lines of code, including comments and blank lines. It compiles into 34,875 words of code and literals. The library adds 5,912 words (including some data area) for a total of 40,787 words. The Univac processor interface routines account for 4,685 words of the library. The data space allocated for the compiler is 16,108 words, and while compiling itself the compiler uses 8,068 words in the heap and 7,444 words in the stack.

The compilation rate is 105 lines per second while producing an output listing and 118 lines per second without producing a listing.

8.2 COMPILED CODE PERFORMANCE

The compiled code was compared with that generated by the Norwegian University Algol (NUALG) and ASCII FORTRAN (FTN) processors. For both Pascal and NUALG, tests were done both with and without run-time checks. The FORTRAN compiler never generates run-time checks but allows for three different levels of optimization. The normal mode does no optimization, and optional modes provide local and global optimization. The local optimization mode was chosen as the standard of comparison, since the short test programs which were used provide an unusually simple case for the global optimizer. This allows it to perform much better than would be expected for the average program.

The programs used as a basis for comparison were taken from Wirth's paper on the design of a Pascal compiler.* They are all short programs which are easily written in all three languages and thus do not use the expressive power of Pascal. In addition, measurements were taken of the time to call a simple procedure and transmit four value parameters. The results are summarized in tables 8.1 and 8.2.

TABLE 8.1. PROCEDURE CALL TIMES.

	Pascal	NUALG	FORTRAN
Time in microseconds	26.4	108.6	21.9
Relative time	1.21	4.96	1.00

*N. Wirth, "The Design of a Pascal Compiler," Software – Practice and Experience, vol. 1, Oct-Dec. 1971, pp. 309-334.

TABLE 8.2. CODE PERFORMANCE.

		Programs				
Language	Time	PART	PARTNP	SORT	MATMUL	COUNT
Pascal	in microseconds	9.36	1.10	24.62	18.70	4.99
with checks	relative	0.62	1.18	1.37	1.82	0.30
Pascal	in microseconds	9.17	0.99	20.22	14.69	4.69
with no checks	relative	0.61	1.06	1.12	1.43	0.28
NUALG	in microseconds	12.88	3.06	32.92	21.02	12.15
with checks	relative	0.85	3.29	1.83	2.05	0.72
NUALG	in microseconds	12.67	2.95	26.81	17.46	11.13
with no checks	relative	0.84	3.17	1.49	1.70	0.66
FTN	in microseconds	15.10	0.87	18.01	10.27	16.88
	relative	1.00	0.94	1.00	1.00	1.00
FTN	in microseconds	15.10	0.93	18.01	10.26	16.83
local	relative	1.00	1.00	1.00	1.00	1.00
FTN	in microseconds	14.94	0.79	10.56	4.04	16.40
global	relative	0.99	0.85	0.59	0.39	0.97

The programs listed are

PART Compute the additive partitions of a number (30 in this case) and print the results. This uses recursive procedures in Pascal and NUALG and a hand-simulated stack in FORTRAN.

PARTNP The same as PART, but with no output.

SORT Sort an array of 1,000 numbers with a bubble sort.

MATMUL Matrix multiply of two 100 X 100 matrices.

COUNT Count the occurrences of each character in a file and print the number of times each occurs. The file was 124,000 character long.

9 IMPLEMENTATION DETAILS

9.1 FORM OF GENERATED CODE

The Pascal compiler generates code in the standard relocatable format, using the following location counters:

1. Procedure code.
2. Literals.
3. Forward reference links.
4. Working storage.

Working storage is allocated at compile time and includes space for all variables allocated at the global level, plus the reserve specified with the "R" compiler directive. At the moment, this is the entire storage allocation for both the stack and the heap. There are plans under way to remove this restriction and allow the dynamic increase of this space up to 262K. The code generated by the compiler will support this, but the run-time system does not.

Forward references are handled by assigning a location under location counter 3 at the time of the reference. When the reference is resolved, the procedure or label location is entered into a table and at the end of the program code is emitted under location counter 3 to complete the references.

9.2 RUN-TIME ORGANIZATION

9.2.1 MEMORY MANAGEMENT

The allocation of working storage is shown in figure 9.1. The working store is used for both the stack and the heap.

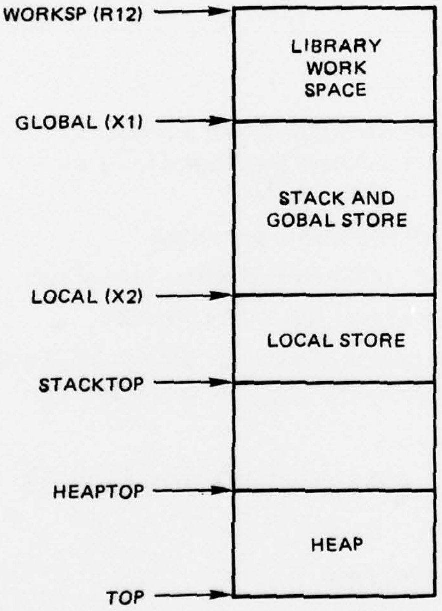


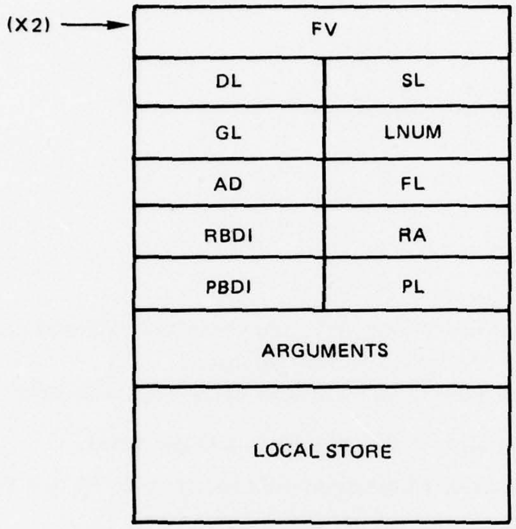
Figure 9.1. Working storage allocation.

Upon entry to an external procedure, X1 is set to point to its own global storage, but local storage for the procedure is put on the same stack as other procedures.

The storage handling on procedure entry and exit is simple and conventional and best seen by examining the library procedures which handle it.

9.2.2 PROCEDURE ENTRY AND EXIT

Each Pascal procedure called has space allocated on the top of the stack for its activation record, which has the form shown in figure 9.2.



FV	– FUNCTION RETURN VALUE
DL	– DYNAMIC LINK (X2 IN SURROUNDING ENV.)
SL	– STATIC LINK (X2 IN STATIC ENV.)
GL	– GLOBAL LINK (X1 IN SURROUNDING ENV.)
LNUM	– LINE NUMBER OF CALL
AD	– ALLOCATION DATA (OLD STACK TOP)
FL	– FILE LINK (FOR CLOSING FILES)
RBDI	– BDI OF THE RETURN ADDRESS
RA	– RETURN ADDRESS
PBDI	– BDI OF THE CALLED PROCEDURE
PL	– PROCEDURE LOCATION FOR DIAG.

Figure 9.2. Activation record.

The procedure calling sequence generated by the compiler is as follows:

```
        SX     "static link",SL
        SX     X2,DL
        .
        "compute parameters"
        .
        AX     X2,"local length",,U   . NEW X2
        LMJ    X11,PROC
        "return location"
```

The code generated in the procedure has the form:

```
        'proc              '       . for diagnostics
PROC    LMJ  X10,P$PENTRY
        +         "argument length","total space"
        "code to store arguments"
        .
        "procedure code"
        .
        LMJ    X10,P$PEXIT
```

The details of filling in the linkage data and for initializing and closing files are best seen by examining the library routines for these functions.

Arguments may be considered to be in four categories as follows:

direct   Those value arguments which will fit in a single word.

indirect   Variable arguments or value arguments too large to fit in a single word.

setvalue   Set arguments passed by value.

proc/func   Formal procedure or function.

In general, arguments are passed in registers whenever possible, with the called procedure storing them in its local activation record. When the number of arguments exceeds the available registers, they are stored in the activation record by the calling program. The compiler allocates space in the activation record as follows:

direct   A single word in the argument area to hold the value.

indirect   A single word in the argument area to hold the address. In addition, value arguments have enough space allocated in the local variable area to hold a local copy of the argument.

setvalue   Sufficient space (4 words currently) to hold the set value.

proc/func   Two words, the first for the instruction and environment pointers, the second for a check word.

The arguments are passed as follows:

direct   Up to 9 arguments are passed in the registers A4 to A12, in that order. Others are passed in core.

indirect Up to 4 arguments are passed in the registers X8 down to X5, in that order. Others have their addresses stored in core.

setvalue Always passed in core.

proc/func The first word is passed as an indirect argument, while the check word is passed as a direct argument.

9.2.3 REGISTER USAGE

The registers X1, X2, and R9 through R15 are used in storage allocation and must be left undisturbed. All other registers are available for use.

9.3 DIAGNOSTIC SYSTEM

The diagnostic system is close to the minimum livable. On error, it provides an error message and a walkback through the stack, listing calling lines and procedure names. In most cases, this provides the data necessary for fault location.

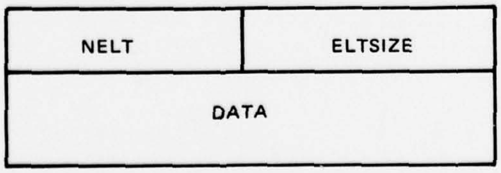
The diagnostic system works by keeping the current line number in the register R10. This line number, plus the address of the called procedure, is kept in the stack on procedure call. At the start of any procedure, the compiler inserts the procedure name in ASCII. This allows a simple trace of the calls and a reasonably readable walkback. For code generated without diagnostic data, the walkback system gives diagnostics in terms of octal locations.

To avoid ambiguities with partially constructed activation records, the address of the latest currently completed mark stack is kept in the register R9.

9.4 REPRESENTATION OF FILES

Text files are represented in the system standard "SDFF" file format. These can be read by the majority of the system processors, such as the editor. The Pascal system can read files produced by any system processor, by FORTRAN-formatted write statements, or by symbionts. If the file is in Fieldata code, it will be translated to ASCII by the file handler. The maximum record length allowed is 132 characters.

Any other data file is written in a system chosen format with a block length which is an integral number of sectors long. The format of each record is shown in figure 9.3.



NELT: NUMBER OF ELTS IN THIS BLOCK

ELTSIZE: IF POSITIVE, ELEMENT SIZE; IF NEGATIVE, ELEMENTS PER WORD

Figure 9.3. Binary file format.

The length of the block, except for the last block in the file, can be computed from nelt*eltsize rounded to the next multiple of 112 words.

The end of file is denoted by a block with nelt <0. On tape, a hardware file mark will follow to allow copying using FURPUR.

9.5 MISCELLANEOUS TOPICS

9.5.1 NON-1110 CODE GENERATION

Pascal 1100 was developed on a Univac 1110, which has slight differences from other computers in the 1100 series. In particular, the real arithmetic on some computers other than the 1110 creates a "residue" stored in the register above the register named in the operation. In order to generate correct code for these machines, the compiler can be instructed via the "E" option to take this residue into account when generating real instructions. To simplify matters for installations without an 1110, there is also a constant "UNIVAC1110" in the compiler, which if false will cause the "E" option to be defaulted to "E+".

The compiler itself does not use any real expressions which will be affected by this option, so that this is strictly a convenience for the final user.

9.5.2 STANDARD FUNCTION REFERENCES

References to the standard functions such as "sin" and "cos" generate a "LIJ" reference to the mathematical function common bank "RMATH$". This is optimum for the Univac 1110, but involves considerable overhead for other members of the 1100 series (approximately 151 microseconds on the 1108.) If the system is installed on a machine other than an 1110, and these functions are expected to see heavy use, it would pay to change the compiler to generate "LMJ" references.